Transferring Information with Files, the Internet and Sockets

Introduction

Reading and writing data to files or transferring data over the Internet are important functions in most applications. LiveCode provides a rich feature set for performing these operations.

Accessing data from a file typically takes just a single line of code. LiveCode's file path syntax uses the same format on each platform so you typically don't have to rewrite your file handling routines to deploy cross platform. A set of functions provides for copying, deleting or renaming files, as well as accessing appropriate system and user folders.

LiveCode includes functions for downloading and uploading data to the Internet. Simple downloads and uploads can be performed with just a single line of code. Support for the http, ftp and post protocols is included. Syntax is included that allows downloading in both the foreground and background. Additional library commands allow you to construct multipart form data, send ftp commands and more.

LiveCode includes built-in support for https, SSL & encryption.

If the built-in protocol support doesn't do what you need, LiveCode also allows you to implement your own Internet protocols using its straightforward socket support. A very basic client server application can be written in a few lines of code.

File Name Specifications and File Paths

A file path is a way of describing the location of a file or folder so that it can be found by a handler. File paths are used throughout LiveCode: when you read to and write from text files, when you reference an external video file to display in a player, and in many other situations. If your application refers to external files in any way, an understanding of file path is essential.

This topic discusses the syntax for creating and reading a file reference, and how to relate file paths to the location of your application so that they'll be accessible when your application is installed on another system with a different folder structure.

What is a File Path?

A file path is a description of the exact location of a file or folder. The file path is created by starting at the top of the computer's file system, naming the disk or volume that the file is on, then naming every folder that encloses the file, in descending order, until the file is reached.

Locating a file

For example, suppose you want to describe the location of a file called "My File", which is located inside a folder called "My Folder". That folder is in turn located inside a folder called "Top Folder", which is on a drive called "Hard Disk". You need all this information to completely describe where the file is: - Hard Disk - Top Folder - My Folder - My File

If someone tells you which disk the file is on, then which folder to open, and so on, you can find the file by opening each successive icon on your computer's desktop. By starting with the disk, then opening each enclosing folder in succession until you arrive at the file, you can find exactly the file that's being described.

The structure of a file path

A file path specifies each level of the hierarchy that encloses the file. LiveCode presents the information in a file path that might look like this:

```
/Hard Disk/Top Folder/My Folder/My File
```

You can see that to write a file path, you start by naming the disk the file is on, then add each enclosing folder in order until you arrive at the file.

To see the path to a file, enter the following in the message box:

```
answer file "Choose a file:"; put it
```

This displays the file path for the file you choose.

> **Important:** Each platform has its own way for programmers to specify file paths. The file path shown above is in the usual style for file paths on Linux systems. For cross-platform compatibility, LiveCode uses this same forward slash / character in its file path regardless of the current platform. This way, you can generally specify file and work with paths in your scripts without having to convert them when you switch platforms.

File paths on Windows systems

On Windows systems, disks are named with a drive letter followed by a colon character (:). A typical LiveCode file path on a Windows system looks like this:

```
C:/folder/file.txt
```

File paths on OS X systems

On OS X systems, the startup disk, rather than the desktop, is used as the top level of the folder structure. This means that the startup disk's name does not appear in file paths. Instead, the first part of the file path is the top-level folder that the file is in.

If the disk "Hard Disk" is the startup disk, a typical path on OS X systems might look like this:

```
/Top Folder/My Folder/My File
```

Notice that the disk name isn't part of this path.

> **Note:** If you need to find out the startup disk's name, check the first disk name returned by the **volumes**function.

For files on a disk that isn't the startup disk, the file path starts with "/Volumes" instead of "/". A typical file path to a file that's on a non-startup disk on an OS X system looks like this:

```
/Volumes/Swap Disk/Folder/file.txt
```

Folder paths

You construct the path of a folder the same way as the path to a file. A folder path always ends with a slash character (/). This final slash indicates that the path is to a folder rather than a file.

For example, this pathname describes a folder called "Project" inside a folder called "Forbin" on a disk named "Doomsday":

```
/Doomsday/Forbin/Project/
```

If "Project" is a file, its pathname looks like this, without the final slash:

```
/Doomsday/Forbin/Project
```

File paths for OS X bundles

A bundle is a special type of folder, used on OS X, that is presented to the user as a single file but that is maintained internally by the operating system as a folder. Many OS X applications – including LiveCode and the applications it creates – are stored and distributed as bundles that contain several files. When the user double-clicks the bundle the application starts up instead of a folder window opening to show the bundle's contents.

You can take advantage of the bundle concept to include any needed support files with your application. If you place the files in the application's bundle, users ordinarily never see them, and the entire application--support files and all--behaves as a single icon.

> **Tip:** To see the contents of a bundle, right-click (or control click) the bundle and choose "Show Package Contents" from the contextual menu.

Most of the time, the distinction between bundles and files doesn't matter. However we recommend that you treat them as files when working from the perspective of a user but otherwise refer to them as folders when coding. This will help to keep your code readable. Thus if you are selecting a bundle in a file dialog use the answer file form. When moving or renaming a bundle, refer to them as a folder.

Moving, renaming, or deleting a bundle

When using the **rename** command, to rename a bundle, use the rename folder form of the command:

```
rename folder "/Volumes/Disk/Applications/MyApp/" to \
    "/Volumes/Disk/Applications/OtherApp/"
```

Similarly, when dealing with a bundle, use the **delete folder** command instead of **delete file**, and the **revCopyFolder** command instead of **revCopyFile**.

Referring to files inside a bundle

When referring to a file that's inside a bundle, you can treat the bundle just as if it were a folder. For example, if you have placed a file called "My Support.txt" inside your application's bundle, the absolute path to the file might look like this:

```
/Volumes/Disk/Applications/MyApp/My Support.txt
```

The / character in a file or folder name

The slash (/) is not a legal character in Unix or Windows file or folder names, but it is legal for Mac OS X file or folder names to contain a slash. Since a slash in a file or folder name would cause ambiguity – is the slash part of a name, or does it separate one level of the hierarchy from the next? – LiveCode substitutes a colon (:) for any slashes in folder or file names on Mac OS X systems.

For example, if a file on a Mac OS X system is named "Notes from 12/21/93", you refer to it in a script as "Notes from 12:21:93". Since the colon is not a legal character in Mac OS X folder or file names, this removes the ambiguity.

Absolute and Relative File Paths

When describing how to get to a file, you have two options. You can start from the top level, the name of the disk, and name each of the enclosing folders until you get to the file. This is called an absolute path, because it's independent of where you start from. Or you can start from the current folder and describe how to get to the file from there. This is called a relative path, because it depends on where you start.

All the file paths shown so far in this topic are absolute paths.

Absolute file paths

Absolute file paths do not depend on which folder your stack file is in or on where the current folder is. An absolute path to a particular folder or file is always written the same way.

For example, suppose your application is in a folder called "Application Folder", and you want to specify a file called "Westwind" which is in a folder called "Stories" inside "Application Folder". - Hard Disk - Top Folder - My Folder - My File - Application Folder - My Application - Stories - Westwind

The absolute file path of your application looks like this:

```
/Hard Disk/Application Folder/My Application
```

and the absolute path of the "Westwind" file looks like this:

```
/Hard Disk/Application Folder/Stories/Westwind
```

Note: On Mac OS X, and Linux systems, absolute file paths always start with a slash character. On Windows systems, absolute file paths always start with a drive letter followed by a colon (:).

Relative file paths

Now suppose you want to tell someone how to get to the "Westwind" file, starting from the folder containing the application.

Since the application is in "Application Folder", we don't need to include the steps to get to "Application Folder". Instead, we can describe the location of the "Westwind" file with this relative pathname:

```
Stories/Westwind
```

This relative pathname starts at "Application Folder"--the folder that holds the application--and describes how to get to the "Westwind" file from there: you open the folder "Stories", then find "Westwind" inside it.

A relative file path starts at a particular folder, rather than at the top of the file system like an absolute file path. The relative file path builds a file path from the starting folder to the file or folder whose location is being specified.

Finding the current folder

By default, the current folder is set to the folder containing the application (either the LiveCode development environment or your application, depending on whether your application is a standalone). So in the example above, the current folder is "Application Folder", because that's where the running application is located.

Note: To change the current folder, set the **defaultFolder** property.

Going up to the parent folder

The relative path ".." indicates the current folder's parent folder. If the current folder is "Stories", the relative path

```
..
```

means the same thing as the absolute path

```
/Hard Disk/Application Folder/
```

Going up multiple levels

To go up more than one level, use more than one "../". To go up two levels, use "../../"; to go up three levels, use "../../../", and so forth.

For example, suppose the current folder is "Stories", and its absolute path looks like this:

```
/Hard Disk/Application Folder/Stories/
```

To get to "My Application" in "Application Folder", you go up one level to "Application Folder", then down one level to "My Application". The relative path looks like this:

```
../My Application
```

To get to "Top Folder" on "Hard Disk", you go up two levels--to "Application Folder", then to "Hard Disk"--and then down one level to "Top Folder". The relative path looks like this:

```
../../Top Folder/
```

Starting at the home directory

On OS X and Unix systems, the "~" character designates a user's home directory.

A path that starts with "~/" is a relative path starting with the current user's home directory. A path that starts with "~", followed by the user ID of a user on that system, is a relative path starting with that user's home directory.

When to use relative and absolute file paths

Absolute file paths and relative file paths are interchangeable. Which one to use depends on a couple of factors.

Absolute file paths are easy to understand and they don't change depending on the current folder. This can be an advantage if you are changing the defaultFolder regularly.

However absolute file paths always include the full name of the hard disk and folders leading up to the current working folder. Therefore, if you plan to distribute your application you will want to work with relative paths, so that media shipped in subfolders with your application is still easy to locate.

> **Tip:** By default, when linking to an image or resource using the Inspector, LiveCode inserts an absolute file path. If you plan to distribute your application, locate your media in a subfolder next to the stack you are working on and convert these file paths to relative file paths by deleting the directories up to the one you are working in. This will mean you don't need to make any changes when it comes time to distribute your application.

It's OK to use absolute paths to specify files or folders that the user selects after installation. For example, if you ask the user to select a file (using the **answer file** command) and read data from the file, there's no need to convert the absolute path that the **answer file** command provides to a relative path. Because you're using the path right after you get it from the **answer** command, you know that the disk name and folder structure aren't going to change between getting the path and using it.

Special Folders

Modern operating systems each have a set of special-purpose folders designated for a variety of purposes. If you are writing an application it is recommended that you make use of these folders where appropriate so that you provide the best possible user experience. For example, the contents of the desktop reside in a special folder; there is a folder set aside for fonts; there is a folder for application preferences; and so on.

These special folders don't always have the same name and location, so you can't rely on a stored file path to locate them. For example, if your application is installed onto an OS localized into a different language, the names of the file path will be different, on some Windows special folders are named or placed differently depending on what version of Windows is running, etc.

To find out the name and location of a special folder, regardless of any of these factors, you use the **specialFolderPath** function. The function supports a number of forms for each operating system, describing the special folders for each one. Some of the forms are the same cross-platform. The following example will get the location of the Desktop folder on Windows, Mac OS X or Linux:

```
put specialFolderPath("Desktop") into myPath
```

To get the path to the Start menu's folder on a Windows system:

```
put specialFolderPath("Start") into myPath
```

For a complete list of possible folders see the, **specialFolderPath** in the *LiveCode Dictionary*.

File Types, Application Signatures & File Ownership

When you double-click a document file, it automatically opens in the application it's associated with. Each operating system has a different method for associating files with an application. In order to create files that belong to your standalone application, you need to set up the association appropriately for each platform you distribute on.

This topic describes how to correctly associate your application with the files it creates.

Windows File Extensions and Ownership

When a file is saved on a Windows system, a three-character extension is usually added to the file's name. The extension specifies the format of the file.

To determine which application to launch when the user double-clicks a file, Windows checks the Windows registry to find out what application has registered itself as owning the file's extension. Each application can add keys to the registry to identify certain file extensions as belonging to it.

Applications that don't own files

If your application does not create files that you want the application to own, you don't need to make any modifications to the registry or specify any extensions.

Applications that own their own files

If your application creates files with its own custom extension, when you install the application, you should make changes to the Windows registry to identify the extension as belonging to your application.

Popular Windows installer programs will make these registry changes automatically for you. You can also perform these registry changes using the **setRegistry** function.

Installing custom icons

Each Windows file can display its own icon. You can have separate icons for your application and for files it owns. Icon files must be stored in .ico format.

Custom application icons

If you want to include a custom icon for your application, use the "Application Icon" option on the Windows screen of the Standalone Application Settings window to specify the icon file. When you build the application, the icon will be included in the application. For more information, see the chapter on *Deploying Your Application*.

Custom file icons

To include a custom icon for your documents, use the "Document Icon" option on the Windows screen of the Standalone Application Settings window to specify the icon file. When you build the application, the icon will be included in the application.

> **Important:** For the correct icon to appear on files your application creates, the file's extension must be registered in the Windows registry.

File extensions

You can add an extension to the name of any Windows file. The extension may contain letters A-Z, digits 0-9, ' (single quote), !, @, #, $, %, ^, &, (,), -, _, {, }, `, or ~.

The Windows registry associates applications with the extension for the files they own.

OS X File Types and Creators

On OS X each file has a file extension which determines which application owns it. However OS X systems can also use the unique four-character creator signature and a four-character file type (see below for more information).

OS X applications store file association information in a property list file, or plist. Each application's plist is stored as part of its application bundle.

Applications that don't own files

To assign your unique creator signature when building an application, enter the signature on the OS X screen of the Standalone Application Settings window. LiveCode automatically includes the creator signature in the application's plist.

Applications that own their own files

If your application creates files with your application's creator signature, you should include in your application's plist an entry for each file type you use. Once you have built your standalone application, follow these steps to open the plist file:

1. Right click on your application bundle, navigate to the contents folder and open the "Info.plist" file. If you have installed Apple's developer tools, you have an application called "Property List Editor", which you can use to make changes to the plist file. Otherwise, you can edit the file in a text editor.

2. Locate the information for the document type. In Property List Editor, expand the "Root" node, then expand the "CFBundleDocumentTypes" node, then expand the "0" node. In a text editor, locate "CFBundleDocumentTypes". Below it, note the tags `<array>` and `<dict>`. The information for the first document type is between `<dict>` and `</dict>`.

3. Enter the file description, which is a short phrase describing what kind of file this is. In Property List Editor, change the value of "CFBundleTypeName" to the description you want to use. In a text editor, locate "CFBundleTypeName" in the document information. Below it is the file description, enclosed between `<string>` and `</string>`:

```
<string>LiveCode Stack</string>
```

Change the description to the one you want to use.

Do not change the tags (enclosed in "\<" and ">"). Only change what's between them.

4. Enter the `file` extension. In Property List Editor, expand "CFBundleTypeExtensions" and enter the `file` extension in the "0" node. In a text editor, locate "CFBundleTypeExtensions" in the document information. Below it is the extension, enclosed in `<array>` and `<string>` tags. Change the extension to the one you want to use.

5. Enter the four-character file type. In Property List Editor, expand "CFBundleTypeOSTypes" and enter the file type in the "0" node. In a text editor, locate "CFBundleTypeOSTypes" in the document information. Below it is the file type, enclosed in `<array>` and `<string>` tags. Change the file type to the one you want to use.

If the format for this type of file is standard (such as plain text), use a standard type (such as "TEXT"). If the format belongs to your application, use a custom file type of your choice.

> **Important:** Apple reserves all file types with no uppercase letters. If you use a custom file type for your application, make sure it contains at least one uppercase letter.

If you want to assign more file types to your application, copy the portion of the plist file inside the "CFBundleTypes" array between `<dict>` and `<dict>`, including these tags. The "CFBundleTypes" node should now contain two `<dict>` nodes and all their contents. Repeat the steps above for each different file type your application can create.

Creating Files

When your application creates files, set the **fileType** property to the desired creator signature and file type for the new file. (For stack files created with the **save** command, use the **stackFileType** property instead.) When creating files, the application uses the current value of the **fileType** or **stackFileType** property to determine what creator and file type the new file should have.

It's important to understand that a file's creator signature determines which application is launched automatically when you double-click the file, but doesn't prevent other applications from being able to open that file. For example, if your application creates files of type "TEXT", any text editor can open the files. If your application creates stack files, and uses the file type "RSTK", then LiveCode will be able to open the stack files, as well as your application.

File extensions

You can add an extension to the name of any OS X file. When the user double-clicks a file with no creator signature, the operating system uses the extension to determine which application to use to open the file.

An application bundle's name should end with the extension ".app".

> **Note:** Apple's recommendations for determining file type and creator on OS X systems are currently in flux. The recommended method for the present is to set a file type and creator signature, and also attach an extension to the end of each file's name when it is created. Valid extensions on OS X systems are up to twelve characters in length, and may include the letters a-z, the digits 0-9, $, %, _, or ~. For up-to-date information on Apple's recommendations for OS X, see Apple's developer documentation.

Mac OS X Classic File Types and Creators

When a file is saved on a Mac OS X system, a four-character creator signature is saved with it. The creator signature specifies which application owns the file. Every Mac OS X application should have a unique creator signature. (Apple maintains a registry of creator signatures on its web site).

Applications that don't own files

To assign your unique creator signature when building an application, enter the signature on the Mac OS X screen of the Standalone Application Settings window. LiveCode automatically includes the resources needed for Mac OS X to recognize the creator signature.

Applications that own their own files

If your application creates files with your application's creator signature, you should include in your application a set of resources for each file type you use. Once you have saved your standalone application, open the application file in ResEdit and follow these steps:

1. Open the BNDL resource window, then open the BNDL 128 resource. The BNDL 128 resource contains a single entry ("APPL").

2. Choose "Create New File Type" from the Resources menu. A new entry appears below the "APPL" entry.

3. In the Type field, enter the four-character file type. If the format for this type of file is standard (such as plain text), use a standard type (such as "TEXT"). If the format belongs to your application, use a custom file type of your choice.

Repeat steps 2-3 for each different file type your application can create.

When your application creates files, set the **fileType** property to the desired creator signature and file type for the new file. For stack files created with the save command, use the **stackFileType** property instead. When creating files, the application uses the current value of the **fileType** or **stackFileType** property to determine what creator and file type the new file should have.

Installing custom icons

Each Mac OS `file` may display any of six different icons, depending on context and on the number of colors the screen can display: large (32x32 pixel) icons and small (16x16 pixel) icons, each in black-and-white, 16 colors, and 256 colors.

Mac OS provides default icons that are used for applications and documents that don't have their own. If you want your application or the documents it owns to display a custom icon, you must create the icons and then attach them to the application.

Custom application icons

If you want to include a custom icon for your application, use ResEdit or a similar tool to create a set of icon resources. There are six standard icon resource types: ICN# (black-and-white), icl4 (four-bit color), icl8 (8-bit color), ics# (black-and-white small), ics4 (4-bit small), and ics8 (8-bit small). Each of these application icons should have the resource ID 128.

Save the icons in a single file, and use the "Include resources from file" option on the Mac OS X screen of the Standalone Application Settings window to specify the file. When you build the application, the icons will be included in the application's file.

Linux File Extensions

Linux systems do not have an overall required method for specifying a file's type, but most files on a Linux system are created with extensions in the file name, similar to the extensions used on Windows systems. These extensions may be of any length and may include any characters (other than /).

Working with URLs

A URL is a container for a file (or other resource), which may either be on the same system the application is running on, or on another system that's accessible via the Internet.

This topic discusses the various URL schemes that LiveCode implements, how to create and manipulate files using URLs, and how to transfer data between your system and an FTP or HTTP server.

To fully understand this topic, you should know how to create objects and write short scripts, and understand how to use variables to hold data. You should also have a basic understanding of how the Internet works.

An Overview of URLs

In the LiveCode language, a URL is a container for a `file` or other document, such as the output of a CGI on a web server. The data in a URL may be on the same system the application is running on, or may be on another system.

URLs in LiveCode are written like the URLs you see in a browser. You use the **URL** keyword to designate a URL, enclosing the URL's name in double quotes:

```
put field "Info" into URL "file:myfile.txt"
get URL "http://www.example.org/stuff/nonsense.html"
put URL "ftp://ftp.example.net/myfile" into field "Data"
```

URL Schemes

A URL scheme is a type of URL. LiveCode supports five URL schemes with the **URL** keyword: **http**, **ftp**, **file**, **binfile**, and (for backwards compatibility on Mac OS X) **resfile**.

The **http** and **ftp** schemes designate documents or directories that are located on another system that's accessible via the Internet. The **file**, **binfile**, and **resfile** schemes designate local files.

The http scheme

An **http** URL designates a document from a web server:

```
put URL "http://www.example.org/home.htm" into field "Page"
```

When you use an `http` URL in an expression, LiveCode downloads the URL from the server and substitutes the downloaded data for the URL.

When you put something into an `http` URL, LiveCode uploads the data to the web server:

```
put field "Info" into URL "http://www.example.net/info.htm"
```

> **Note:** Because most web servers do not allow `http` uploads, putting something into an `http` URL usually will not be successful. Check with the server's administrator to find out whether you can use the `http` protocol to upload files.

For more details about `http` URLs, see the entry for the `http` keyword in the LiveCode Dictionary.

The ftp scheme

An **ftp** URL designates a file or directory on an FTP server:

```
get URL "ftp://user:passwd@ftp.example.net/picture.jpg"
```

When you use an `ftp` URL in an expression, LiveCode downloads the URL from the server and substitutes the downloaded data for the URL. When you put something into an ftp URL, LiveCode uploads the data to the ftp server:

```
put image 10 into URL \
    "ftp://user:passwd@ftp.example.net/picture.jpg"
```

FTP servers require a user name and password, which you can specify in the URL. If you don't specify a user name and password, LiveCode adds the "anonymous" user name and a dummy password automatically, in accordance with the convention for public FTP servers.

Note: Uploading to an FTP server usually requires a registered user name and password.

For more details about `ftp` URLs, see the entry for the `ftp` keyword in the LiveCode Dictionary.

Directories on an FTP server

A URL that ends with a slash (/) designates a directory (rather than a file). An `ftp` URL to a directory evaluates to a listing of the directory's contents.

The file scheme

A **file** URL designates a file on your system:

```
put field "Stuff" into URL "file:/Disk/Folder/testfile"
```

When you use a `file` URL in an expression, LiveCode gets the contents of the `file` you designate and substitutes it for the URL. The following example puts the contents of a `file` into a variable:

```
put URL "file:myfile.txt" into myVariable
```

When you put data into a `file` URL, LiveCode puts the data into the file:

```
put myVariable into URL "file:/Volumes/Backup/data"
```

Note: As with local variables, if the file doesn't exist, putting data into it creates the file.

To create a URL from a file path that LiveCode provides, use the **&** operator:

```
answer file "Please choose a file to get:"
get URL ("file:" & it)
```

File path syntax and the file scheme:

The `file` URL scheme uses the same `file` path syntax used elsewhere in LiveCode statements. You can use both absolute paths and relative paths in a `file` URL.

Conversion of end-of-line markers

Different operating systems use different characters to mark the end of a line. Mac OS X uses a return character (ASCII 13), Linux systems use a linefeed character (ASCII 10), and Windows systems use a return followed by a linefeed. To avoid problems when transporting a stack between platforms, LiveCode always uses linefeeds internally when you use a `file` URL as a container. LiveCode translates as needed between the your system's end-of-line marker and LiveCode's linefeed character. To avoid this translation, use the `binfile` scheme (see below).

The binfile scheme

A **binfile** URL designates a file on your system that contains binary data:

```
put URL "binfile:beachball.gif" into image "Beachball"
```

When you use a **binfile** URL in an expression, LiveCode gets the contents of the file you designate and substitutes it for the URL. The following example puts the contents of a file into a variable:

```
put URL "binfile:picture.png" into pictVar
```

When you put data into a **binfile** URL, LiveCode puts the data into the file:

```
put pictVar into URL "binfile:/Volumes/Backup/pict.png"
put image 1 into "binfile:/image.png"
```

As with local variables, if the file doesn't exist, putting data into it creates the file.

The **binfile** scheme works like the file scheme, except that LiveCode does not attempt to convert end-of-line markers. This is because return and linefeed characters can be present in a binary file but not be intended to mark the end of the line. Changing these characters can corrupt a binary file, so the **binfile** scheme leaves them alone.

The resfile scheme

On Mac OS Classic (and sometimes on OS X systems), files can consist of either a data fork or a resource fork or both.

> **Important:** While LiveCode supports reading and writing resource fork files on Mac OS X, this feature is only intended to help you access and work with legacy files. We do not generally recommend the use of resource forks when designing any new application.

The resource fork contains defined resources such as icons, menu definitions, dialog boxes, fonts, and so forth. A **resfile** URL designates the resource fork of a Mac OS X file:

```
put myBinaryData into URL "resfile:/Disk/Resources"
```

When you use a **resfile** URL in an expression, LiveCode gets the resource fork of the file you designate and substitutes it for the URL.

When you put data into a **resfile** URL, LiveCode puts the data into the file's resource fork.

> **Note:** A **resfile** URL specifies the entire resource fork, not just one resource. To work with individual resources, use the **getResource**, **setResource**, **deleteResource** and **copyResource** functions.

The most common use for this URL scheme is to copy an entire resource fork from one file to another. To modify the data from a **resfile** URL, you need to understand the details of Apple's resource fork format.

Creating a resource fork

Unlike the **file** and **binfile** URL schemes, the **resfile** keyword cannot be used to create a file. If the file doesn't yet exist, you cannot use the **resfile** keyword to create it. To create a new resource file, first use a **file** URL to create the file with an empty data fork, then write the needed data to its resource fork:

```
put empty into URL "file:myFile" -- creates an empty file
put myStoredResources into URL "resfile:myFile"
```

Manipulating URL contents

You use a URL like any other container. You can get the content of a URL or use its content in any expression. You can also put any data into a URL.

http, **ftp**, **binfile**, and **resfile** URLs can hold binary data.

http, **ftp**, and **file** URLs can hold text.

The URL keyword

To specify a URL container, you use the **URL** keyword before the URL, which can use any of the five schemes described above:

```
if URL "http://www.example.net/index.html" is not empty then ...

get URL "binfile:/Applications/Hover.app/data"

put 1+1 into URL "file:output.txt"
```

The `URL` keyword tells LiveCode that you are using the `URL` as a container.

Some properties (such as the **filename** of a player or image) let you specify a URL as the property's value. Be careful not to include the **URL** keyword when specifying such properties, because using the URLkeyword indicates that you're treating the URL as a container. If you use the URL keyword when specifying such a property, the property is set to the contents of the URL, not the URL itself, and this is usually not what's wanted.

Using the content of a URL

As with other containers, you use the content of a URL by using a reference to the URL in an expression. LiveCode substitutes the URL's content for the reference.

If the URL scheme refers to a local file (**file**, **binfile**, or **resfile** URLs), LiveCode reads the content of the file and substitutes it for the URL reference in the expression:

```
answer URL "file:../My File"
-- displays the file's content
put URL "binfile:flowers.jpg" into myVariable
put URL "resfile:Icons" into URL "resfile:New Icons"
```

If the URL scheme refers to a document on another system (**http** or **ftp** URLs), LiveCode downloads the URL automatically, substituting the downloaded data for the URL reference:

```
answer URL "http://www.example.net/files/greeting.txt"
```

Note: If the server sends back an error message--for example, if the file you specify in an **http** URL doesn't exist--then the error message replaces the URL reference in the expression.

Important: When you use an **ftp** or **http** URL in an expression, the handler pauses until LiveCode is finished downloading the URL. If you do not want to block LiveCode when accessing these resources, use the **load URL** form of the command (see below).

Putting data into a URL

As with other containers, you can put data into a URL. The result of doing so depends on whether the URL scheme specifies a file on your system (**file**, **binfile**, or **resfile**) or on another system (**http** or **ftp**).

If the URL scheme refers to a local file (**file**, **binfile**, or **resfile** URLs), LiveCode puts the data into the specified file:

```
put field "My Text" into URL "file:storedtext.txt"
put image 1 into URL "binfile:picture.png"
```

If the URL scheme refers to a document on the Internet (**http** or **ftp** URLs), LiveCode uploads the data to the URL:

```
put myVar into URL "ftp://me:pass@ftp.example.net/file.dat"
```

Because most web servers do not allow HTTP uploads, this usually will not be successful with the **http**scheme.

Chunk expressions and URLs

Like other containers, URLs can be used with chunk expressions to specify a portion of what's in a URL--a line, an item, a word, or a character. In this way, any chunk of a URL is like a container itself. For more information about Chunk Expressions, see the guide on *Processing Text and Data*.

You can use any chunk of a URL in an expression, in the same way you use a whole URL:

```
get line 2 of URL "http://www.example.net/index.html"
put word 8 of URL "file:/Disk/Folder/myfile" into field 4
if char 1 of URL "ftp://ftp.example.org/test.jpg" is "0" then ...
```

You can also specify ranges, and even one chunk inside another:

```
put char 1 to 30 of URL "binfile:/marks.dat" into myVar
answer line 1 to 3 of URL "http://www.example.com/file"
```

Putting data into a chunk

If the URL is local (that is, if it is a **file**, **binfile**, or **resfile** URL), you can put a value into a chunk of the URL:

```
put it into char 7 of URL "binfile:/picture.gif" put return after \
     word 25 of URL "file:../datafile"
put field 3 into line 20 of URL "file:myfile.txt"
```

You can also put a value into a chunk of an **ftp** or **http** URL. Because it's impossible to upload part of a file, LiveCode downloads the file, makes the change, then uploads the file back to the server.

> **Tip:** This method is inefficient if you need to make several changes. In this case, it's faster to first put the URL in a variable, replace the chunk you want to change, then put the variable into the URL:

```
put URL "ftp://me:secret@ftp.example.net/file.txt" into myVar
put field "New Info" after line 7 of myVar
put field "More" into word 22 of line 3 of myVar
put myVar into URL "ftp://me:secret@ftp.example.net/file.txt"
```

This ensures that the file only needs to be downloaded once and re-uploaded once, no matter how many changes you need to make.

URLs and memory

URLs, unlike other containers, are only read into memory when you use the URL in a statement. Other containers – like variables, fields, buttons, and images – are normally kept in memory, so accessing them doesn't increase memory usage.

This means that in order to read a URL or place a value in a chunk of a URL, LiveCode reads the entire file into memory. Because of this, you should be cautious when using a URL to refer to any very large file.

Even when referring to a single chunk in a URL, LiveCode must place the entire URL in memory. An expression such as line 347882 of URL "file:bigfile.txt" may be evaluated very slowly or even not work at all, if insufficient memory is available. If you refer to a chunk of an `ftp` or http URL, LiveCode must download the entire file to find the chunk you specify.

If you need to read and write large quantities of data to a file, or seek through the contents of a large file without loading the entire contents into memory, use the **open file**, **read from file**, **seek** and **close file**commands instead of the URL commands. For more information on these commands see the *LiveCode Dictionary*.

Deleting URLs

You remove a URL with the **delete URL** command.

To delete a local file, you use a **file** or **binfile** URL:

```
delete URL "file:C:/My Programs/test.exe"
delete URL"binfile:../mytext.txt"
```

It doesn't matter whether the file contains binary data or text; for deletion, these URL schemes are equivalent.

> **Tip:** You can also use the **delete file** command to remove a file. To delete the resource fork of a file, you use a **resfile** URL. The following example removes the resource fork along with all resources, but leaves the file in place:

```
delete URL "resfile:/Volumes/Backup/proj.rev"
```

> **Tip:** To delete a single resource instead of the entire resource fork, use the **deleteResource** function.

To remove a file or directory from an FTP server, you use an **ftp** URL:

```
delete URL "ftp://root:secret@ftp.example.org/deleteme.txt"
delete URL "ftp://me:mine@ftp.example.net/trash/"
```

As with creating files, you can use an **http** URL to delete a file, but most HTTP servers are not configured to allow this.

Uploading and Downloading Files

The simplest way to transfer data to an FTP or HTTP server is to use the **put** command to upload, or use the URL in an expression to download.

The Internet library includes additional commands to upload and download files to and from an FTP server. These commands offer more versatile options for monitoring and controlling the progress of the file transfer.

Uploading using the put command

As mentioned above, putting something into an **ftp** or **http** URL uploads the data to the server:

```
put myVariable into URL
"ftp://user:pass@ftp.example.org/newfile.txt"
```

If you use the **put** command with a **file** or **binfile** URL as the source, the file is uploaded:

```
put URL "file:newfile.txt" into URL
"ftp://user:pass@ftp.example.org/newfile.txt"
```

When you upload data in this way, the operation is blocking: that is, the handler pauses until the upload is finished. (See below for details on how to create a file transfer that is not blocking.) If there is an error, the error is placed in the **result** function:

```
put field "Data" into URL myFTPDestination
if the result is not empty then beep 2
```

Important: Uploading or downloading a URL does not prevent other messages from being sent during the file transfer: the current handler is blocked, but other handlers are not. For example, the user might click a button that uploads or downloads another URL while the first URL is still being uploaded. In this case, the second file transfer is not performed and the **result** is set to "Error Previous request has not completed." To avoid this problem, you can set a flag while a URL is being uploaded, and check that flag when trying to upload or download URLs to make sure that there is not already a file transfer in progress.

Downloading using a URL

Referring to an **ftp** or **http** URL in an expression downloads the document.

```
put URL "ftp://ftp.example.net/myfile.jpg" into image 1
get URL "http://www.example.com/newstuff/newfile.html"
```

If you use the **put** command with a **file** or **binfile** URL as the destination, the document is downloaded to the file:

```
put URL "ftp://ftp.example.net/myfile.jpg" into URL \
    "binfile:/Disk/Folder/myfile.jpg"
```

Non-blocking transfers

When you transfer a file using URL containers, the file transfer stops the current handler until the transfer is done. This kind of operation is called a blocking operation, since it blocks the current handler as long as it's going on.

If you want to transfer data using *http* without blocking, use the **load** command. if you want to transfer large files using *ftp*, use the **libURLftpUpload**, **libURLftpUploadFile**, or **libURLDownloadToFile** commands.

Non-blocking file transfers have several advantages:

Since contacting a server may take some time due to network lag, the pause involved in a blocking operation may be long enough to be noticeable to the user.

If a blocking operation involving a URL is going on, no other blocking operation can start until the previous one is finished. If a non-blocking file transfer is going on, however, you can start other non-blocking file transfers. This means that if you use the library commands, the user can begin multiple file transfers without errors.

During a non-blocking file transfer, you can check and display the status of the transfer. This lets you display the transfer's progress and allow the user to cancel the file transfer.

Using the load command

The **load** command downloads the specified document in the background and places it in a cache. Once a document has been cached, it can be accessed nearly instantaneously when you use its URL, because LiveCode uses the cached copy in memory instead of downloading the URL again.

To use a file that has been downloaded by the load command, refer to it using the URL keyword as usual. When you request the original URL, LiveCode uses the cached file automatically.

For best performance, use the `load` command at a time when response speed isn't critical (such as when your application is starting up), and only use it for documents that must be displayed quickly, such as images from the web that will be shown when you go to the next card.

Checking status when using the load command

While a file is being transferred using the load commands, you can check the status of the transfer using the **URLStatus** function. This function returns the current status of a URL that's being downloaded or uploaded:

```
local tUrl
put "ftp://ftp.example.com/myfile.txt" into tUrl
put the URLStatus of tUrl into field "Current Status"
```

The **URLStatus** function returns one of the following values:

- *queued* : on hold until a previous request to the same site is completed
- *contacted* : the site has been contacted but no data has been sent or received yet
- *requested* : the URL has been requested
- *loading bytesTotal*, *bytesReceived* : the URL data is being received
- *uploading bytesTotal, bytesReceived* : the file is being uploaded to the URL
- *cached* : the URL is in the cache and the download is complete
- *uploaded* : the application has finished uploading the file to the URL
- *error* : an error occurred and the URL was not transferred
- *timeout* : the application timed out when attempting to transfer the URL

To monitor the progress of a file transfer or display a progress bar, you check the **URLStatus** function repeatedly during the transfer. The easiest way to do this is with timer based messaging – see the section of the same name in the *LiveCode Script* guide, for more information.

Canceling a file transfer & emptying the cache

To cancel a transfer initiated with the load command and empty the cache, use the **unload** command.

```
unload URL "http://example.org/new\_beta"
```

Uploading and downloading large files using FTP

The Internet library provides a number of commands for transferring larger files via FTP without blocking.

- **libURLftpUpload** uploads data to an FTP server
- **libURLftpUploadFile** uploads a file to an FTP server
- **libURLDownloadToFile** downloads a file from an FTP server to a local file

The basic effect of these commands is the same as the effect of using URLs: that is, the data is transferred to or from the server. However, there are several differences in how the actual file transfer is handled. Because of these differences, the library commands are more suitable for uploads and downloads, particularly if the file being transferred is large.

The following sets of statements each show one of the Internet library commands, with the equivalent use of a URL:

```
libURLftpUpload myVar,"ftp://me:pass@example.net/file.txt"
put myVar into URL "ftp://me:pass@example.net/file.txt"

libURLftpUploadFile "test.data","ftp://ftp.example.org/test"
put URL "binfile:test.data" into URL "ftp://ftp.example.org/test

libURLDownloadToFile "ftp://example.org/new\_beta","/HD/File"
put URL "ftp://example.org/new\_beta" into URL "binfile:/HD/File"
```

Using callback messages

When you start a file transfer using the **libURLftpUpload**, **libURLftpUploadFile**, or **libURLDownloadToFile**command, you can optionally specify a callback message, which is usually a custom message that you write a handler for. This message is sent whenever the file transfer's **URLStatus** changes, so you can handle the callback message to handle errors or to display the file transfer's status to the user.

The following simple example demonstrates how to display a status message to the user. The following handlers might be found in a button's script:

```
on mouseUp
    local tUrl
    put "ftp://example.org/new_beta" into tUrl
    libURLDownloadToFile tUrl,"/HD/Latest Beta","showStatus"
end mouseUp

on showStatus theURL
    put the URLStatus of theURL into field "Status"
end showStatus
```

When you click the button, the **mouseUp** handler is executed. The **libURLDownloadToFile** command begins the file transfer, and its last parameter specifies that a *showStatus* message will be sent to the button whenever the **URLStatus** changes.

As the **URLStatus** changes periodically throughout the download process, the button's *showStatus* handler is executed repeatedly. Each time a *showStatus* message is sent, the handler places the new status in a field. The user can check this field at any time during the file transfer to see whether the download has started, how much of the file has been transferred, and whether there has been an error.

If a file transfer was started using the **libURLftpUpload**, **libURLftpUploadFile**, or **libURLDownloadToFile**command, you can cancel the transfer using the **unload** command.

Uploading, downloading, and memory

When you use a URL as a container, LiveCode places the entire URL in memory. For example, if you download a file from an FTP server using the **put** command, LiveCode downloads the whole contents of the file into memory before putting it into the destination container. If the file is too large to fit into available memory, a file transfer using this method will fail (and may cause other unexpected results).

The library commands **libURLftpUpload**, **libURLftpUploadFile**, and **libURLDownloadToFile**, however, do not require the entire file to be loaded into memory. Instead, they transfer the file one piece at a time. If a file is (or might be) too large to comfortably fit into available memory, you should always use the library commands to transfer it.

Using a stack on a server

Ordinarily, you use stack files that are located on a local disk. You can also open and use a stack that is located on an FTP or HTTP server. Using this capability, you can update an application by downloading new stacks, make new functionality available via the Internet, and even keep most of your application on a server instead of storing it locally.

Going to a stack on a server:

As with local stack files, you use the **go** command to open a stack that's stored on a server:

```
go stack URL "http://www.example.org/myapp/main.rev"
go stack URL "ftp://user:pass@example.net/secret.rev"
```

Note: For such a statement to work, the stack file must have been uploaded as binary data, uncompressed, and not use encodings such as BinHex.

Tip: If you need to download a large stack, use the **load** command to complete the download before using the **go** command to display the stack. This allows you to display a progress bar during the download.

LiveCode automatically downloads the stack file. The main stack of the stack file then opens in a window, just as though you had used the **go** command to open a local stack file.

You can go directly to a specific card in the stack:

```
local tStackUrl
put "http://www.example.org/myapp/main.rev" into tStackUrl
go card "My Card" of stack URL tStackUrl
```

To open a substack instead, use the substack's name:

```
local tStackUrl
put "http://www.example.org/myapp/main.rev" into tStackUrl
go stack "My Substack" of stack URL tStackUrl
```

Using a compressed stack

You cannot directly open a stack that's compressed. However, since the stack URL is a container, you can use the URL as the parameter for the **decompress** function. The function takes the stack file data and decompresses it, producing the data of the original stack file. You can open the output of the function directly as a stack.

The following statement opens a compressed stack file on a server:

```
go decompress(stack URL "http://www.example.net/comp.gz")
```

The statement automatically downloads the file "comp.gz", uncompresses it, and opens the main stack of the file.

Saving stacks from a server

When a stack is downloaded using the **go** command, it's loaded into memory, but not saved on a local disk. Such a stack behaves like a new (unsaved) stack until you use the **save** command to save it as a stack file.

> **Note:** Saving a stack that has been downloaded with the **go** command does not re-upload it to its server. To upload a changed stack, you must save it to a local file, then use one of the methods described in this topic to upload the file to the server.

Other Internet Commands

The Internet library has a number of additional commands for working with web forms, ftp commands, custom settings and troubleshooting. These commands are documented in more detail the LiveCode Dictionary.

Launching the User's Browser with a URL

To launch the default browser with a URL, use the **launch URL** command.

```
launch URL "http://www.livecode.com/"
```

> **Note:** To render web pages within LiveCode, instead of launching an external browser, use the revBrowser. See the section on revBrowser for more information.

Working with Web Forms

To post data to a web form, use the **post** command. To encode data to make it suitable for posting, use the **libUrlFormData** function. To create multi-part form data (as described in RFC 1867) use the**libUrlMultipartFormData** function. To add data to a multipart form one part at a time, use the **libUrlMultipartFormAddPart** function. This can be useful if you need to specify the mime type or transfer encoding for each part.

Working with FTP

For details on basic uploading and downloading using FTP, see the section above.

The following commands provide additional capabilities when working with the ftp protocol:

- **libURLSetFTPStopTime** : Sets the timeout value for FTP transfers.
- **libURLSetFTPMode** : Switches between active and passive mode for FTP transfers.
- **libURLSetFTPListCommand** : Switches between sending LIST or NLST formats when listing the contents of an FTP directory.
- **libURLftpCommand** : sends an ftp command to an ftp server.
- **libURLftpUpload** : uploads data. See the section above for more details.
- **libURLftpUploadFile** : uploads a file, without loading the entire file into memory. See the section above for more details.
- **libURLDownloadToFile** – downloads data to a file, without loading the entire data into memory. See the section above for more details.

HTTP methods and http URLs

The basic operations used by the HTTP protocol are called methods. For **http** URLs, the following HTTP methods are used under the following circumstances:

- GET: when an **http** URL in an expression is evaluated
- PUT: when you put a value into an **http** URL
- POST: when you use the **post** command
- DELETE: when you use the **delete URL** command with an **http** URL

> **Note:** Many HTTP servers do not implement the PUT and DELETE methods, which means that you can't put values into an **http** URL or delete an **http** URL on such servers. It's common to use the FTP protocol instead to upload and delete files; check with your server's administrator to find out what methods are supported.

HTTP headers

When LiveCode issues a GET or POST request, it constructs a minimal set of HTTP headers. For example, when issued on a Mac OS system, the statement:

```
put URL "http://www.example.org/myfile" into myVariable
```

results in sending a GET request to the server:

```
GET /myfile HTTP/1.1 Host: 127.0.0.0 User-Agent: LiveCode (MacOS)
```

You can add headers, or replace the Host or User-Agent header, by setting the **HTTPHeaders** property before using the URL:

```
set the HTTPHeaders to "User-Agent: MyApp" & return \
    & "Connection: close"
put URL "http://www.example.org/myfile" into myVariable
```

Now the request sent to the server looks like this:

GET /myfile HTTP/1.1 Host: 127.0.0.0 User-Agent: MyApp Connection: close

The **ftp** URL scheme can be used to create a new file to an FTP server. As with the **file** and **binfile** schemes, putting something into the URL creates the file:

```
put dataToUpload into URL
"ftp://jane:pass@ftp.example.com/newfile.dat"
```

> **Tip:** You can create an FTP directory by uploading a file to the new (nonexistent) directory. The directory is automatically created. You can then delete the file, if you wish, leaving a new, empty directory on the server:

```
-- Create an empty file in the nonexistent directory:
put empty into URL "ftp://jane:pass@example.com/newdir/dummy"

-- Delete unwanted empty file to leave new directory:
delete URL "ftp://jane:pass@example.com/newdir/dummy"
```

Additional Transmission Settings

The following commands provide additional customization options for the Internet library:

- **libUrlSetExpect100** : Allows you to set a limit to the size of data being posted before requesting a continue response from the server.
- **libURLSetCustomHTTPHeaders** : Sets `the header`s to be sent with each request to an HTTP server. See also the section on HTTPHeaders above.
- **libURLFollowHttpRedirects** : Specify that GET requests should follow HTTP redirects and GET the page redirected to.
- **libUrlSetAuthCallback** : Sets a callback for handling authentication with http servers and proxies.

Troubleshooting

The following commands and functions can be useful when debugging an application that uses the Internet library.

- **resetAll** : Closes all open sockets and halts all pending Internet operations.

> **Caution:** The **resetAll** command closes all open sockets, which includes any other sockets opened by your application and any sockets in use for other uploads and downloads. Because of this, you should avoid routine use of the **resetAll** command. Consider using it only during development, to clear up connection problems during debugging.

- **libURLErrorData** : Returns any error that was caused during a download that was started with the load command.
- **libURLVersion** : Returns the version of the Internet library.
- **libURLSetLogField** : Specifies a field for logging information about uploads and downloads on screen.
- **libURLLastRHHeaders** : Returns the headers sent by the remote host in the most recent HTTP transaction.
- **libURLLastHTTPHeaders** : Returns the value of the httpHeadersproperty used for the previous HTTP request.

revBrowser – Rendering a Web Page within a Stack

Use the revBrowser commands to render a web page within a stack. RevBrowser uses WebKit (Safari) on Mac OS X and Internet Explorer on Windows. Currently RevBrowser is not supported under Linux.

To create a browser object in a stack, use the **revBrowserOpen** function. This function takes the **windowID**for the stack you want to open the browser in and a URL. Please note that the `windowID` is not the same as the stack's ID property.

```
put the windowid of this stack into tWinID
put revBrowserOpen(tWinID,"http://www.google.com") into sBrowserId
```

To set properties on the browser, use the **revBrowserSet** command. The following commands makes the border visible then sets the rectangle to be the same as an image named "browserimage":

```
revBrowserSet sBrowserId, "showborder","true"
revBrowserSet sBrowserId, "rect",rect of img "browserimage"
```

To close a browser when you finished with it, use the **revBrowserClose** command. This command takes the `windowID` for the stack containing the browser:

```
revBrowserClose sBrowserId
```

RevBrowser supports a number of settings and messages. You can intercept a message whenever the user navigates to a link, prevent navigation, intercept clicks in the browser, requests to download files or to open a new window.

For a complete list of commands that operate on RevBrowser, open the *LiveCode Dictionary* and type "browser" into the filter box.

SSL and Encryption

LiveCode includes support for using Secure Sockets Layer and the https protocol. It also includes an industrial strength encryption library you can use to encrypt files or data transmissions.

Encrypting and Decrypting Data

To encrypt data, use the **encrypt** command. The **encrypt** command supports a wide variety of industry standard methods of encryption. The list of installed methods can be retrieved by using the **cipherNames**function. To decrypt data, use the **decrypt** command. For more information on these features, see the *LiveCode Dictionary*.

> **Tip:** If you are using the encryption library on a Windows system, it is possible that another application will have installed DLLs that use the same name as the ones included with LiveCode to support encryption. You can force your application to load LiveCode's SSL DLLs by setting the $PATH environment variable before loading the library.

```
put $PATH into tOldPath
put <path to SSL DLLs> into $PATH
get the cipherNames -- Force loading of the SSL DLLs
put tOldPath into $PATH
```

Connecting using HTTPS

You may connect and download data from a URL using **https** in the same way that you access an http URL.

```
put URL "https://www.example.com/store.php"
```

If there is an error, it will be placed into `the result`. If you need to include a user name and password you can do so in the following form:

```
https://user:password@www.example.com/
```

Implementing your own secure protocols

To implement your own secure protocol, use the **open secure socket** variant of the **open socket** command. You can specify whether or not to include certification, a certificate and a key. For more information on the **open socket** command, see the *LiveCode Dictionary*.

Writing your own protocol with sockets

If you need to implement your own protocol, you can do so using LiveCode's socket support. To understand this chapter it is assumed you understand the basics of how the Internet works, including the concepts of sockets, IP addresses and ports. More information on these concepts can be found in Wikipedia.

> **Tip:** The standard protocols that LiveCode support such as http and ftp, discussed earlier in this chapter, have all been implemented as a scripted library with LiveCode's socket support. You can examine this library by running `edit script of stack "revlibURL"` in the Message Box. Beware, this library is not for the faint of heart. If you change anything, LiveCode's Internet commands may cease to operate.

Opening a connection

To open a connection use the **open socket** command. The following command opens a connection to the IP address specified in the `tIPAddress` variable and the port specified in the `tPort` variable. It specifies that LiveCode should send the message "`chatConnected`" when a connection has been established.

```
open socket (tIPAddress & ":" & tPort) with message "chatConnected"
```

To open a secure socket, use the **open secure socket** variant of the command. To open a UDP datagram socket, use the **open datagram socket** variant of the command. For more information on these variants, see the *LiveCode Dictionary*.

Looking up a host name or IP address

You may look up an IP address from a host name with the **hostNameToAddress** function. For example, to get the IP address for the livecode.com server:

```
put hostNameToAddress("www.livecode.com") into tIPAddress
```

To get the host name of the local machine, use the **hostName** function. To look up the name from an IP address, use the **hostAddressToName** function.

Reading and writing data

Once LiveCode opens a connection, it will send a `chatConnected` message. To receive data, use the **read from socket** command. The following message reads data from the socket and sends a `chatReceived` message when reading is completed.

```
on chatConnected pSocket
    read from socket pSocket with message chatReceived
end chatConnected
```

Once reading from the socket is completed the `chatReceived` message can be used to process or display the data. It can then specify that it should continue to read from the socket until more data is received, sending another `chatReceived` message when done.

```
on chatReceived pSocket, pData
    put pData after field "chat output"
    read from socket pSocket with message "chatReceived"
end chatReceived
```

To write data to the socket, use the **write** command:

```
write field "chat text" to socket tSocket
```

Disconnecting

To disconnect, use the **close socket** command. You should store a variable with details of any open sockets and close them when you have finished using them or when your stack closes.

```
close socket (tIDAddress & ":" & tPort)
```

Listening for and accepting incoming connections

To accept incoming connections on a given port, use the **accept connections** command. The following example tells LiveCode to listen for connections on port 1987 and send the message `chatConnected` if a connection is established. You can then start to read data from the socket in the `chatConnected` handler.

```
accept connections on port 1987 with message chatConnected
```

Handling errors

If there is an error, LiveCode will send a **socketError** message with the address of the socket and the error message. If a socket is closed a **socketClosed** message will be sent. If a socket times out waiting for data a **socketTimeout** message will be sent. To get a list of sockets that are open, use the **openSockets** function. You can set the default timeout interval by setting the **socketTimeOutInterval** property. For more details on all of these features, see the *LiveCode Dictionary*.

> **Tip:** You can see a complete implementation of a basic client server "chat" application by navigating to Documentation -> Getting Started -> Sample Projects -> Internet Chat – creating a custom protocol using sockets -> Launch. Most of the scripts for the "server" stack are in the "start server" button. Most of the scripts for the client are in the stack script for the "chat client" stack.